简易数据存储系统说明文档

姓名：张云昊

学号：5130309063

目录

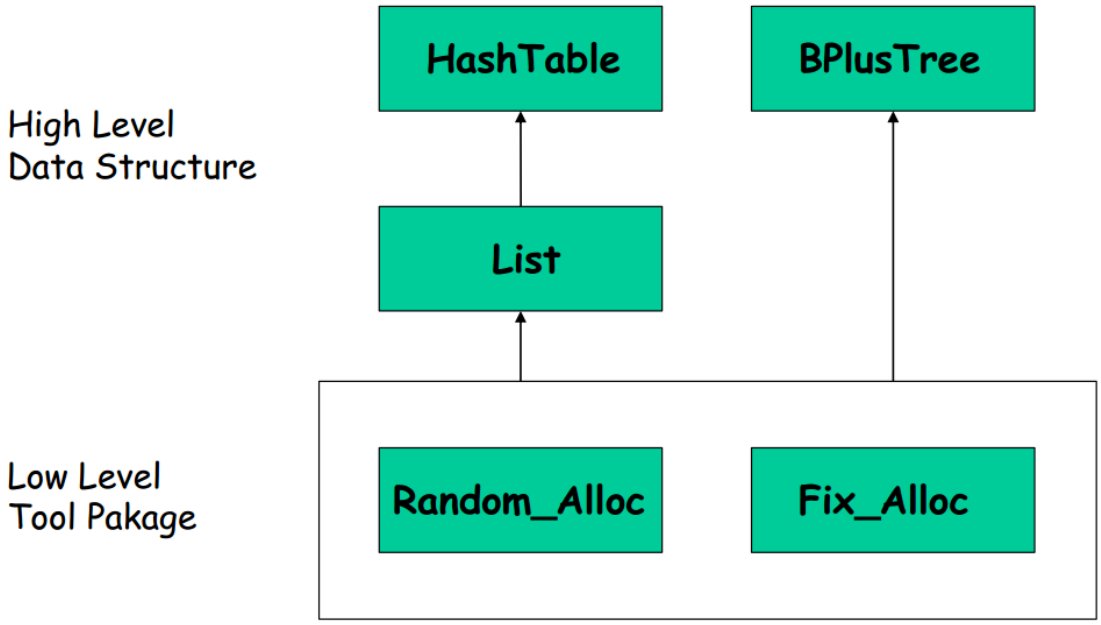
一、 概述

我实现了两个数据存储系统，一个使用哈希表实现，一个使用 B+树实现。为了更好地实现代码复用，我将文件操作的部分单独写成类，给上层数据结构提供了方便的接口，使得操作磁盘就像操作内存。然后我对两个系统的性能进行了测试，测试结果和理论符合的很好，充分的体现了哈希表和 B+树各自的优缺点。

本文档的内容包括：底层数据结构，哈希表实现，B+树实现，测试设计，测试结果，测试结论等。

二、 底层数据结构

整个数据存储系统可以用这个图来表示：



底层数据结构包括两个 class，一个是 Random_Alloc，另一个是 Fix_Alloc。这两个类分别满足了如下两个需求：

1. 存储任意长的字符串，key 和 value 都是普通字符串，所以都会用 Random_Alloc 来存储。要支持插入、删除、**比较大小**等操作。
2. 存储定长的 struct 或者 class。Random_Alloc 会对字符串进行整页对齐，但不论是树还是哈希表，每一个节点或者表项都只需要占用很少的内存(比如我实现的 class List 占用 24byte)，所以可以在一个页里存储多个 class。同时，还需要把一个结构体映射/反映射到虚存中的功能。

同时，作为底层工具，这两个类还要考虑一些底层效率问题，比如页对齐，缓存，mmap 内存映射等。

i. 可变长字符串存储

首先介绍我的 Random_Alloc 实现。一个基本的宏定义是 page_t。如果一个变量 x 的类型是 page_t，那么它代表了文件中的一个位置，即第 x 个页的开始位置。

构造函数：

```
Random_Alloc(const char* filename)
```

输入一个文件名作为操作可变长字符串的容器。

插入字符串：

```
page_t insert(const char *value, int len)
```

插入函数输入一个字符串 value 和它的长度，并将它存储到文件里。返回值是这个字符串现在在文件中的位置。

删除字符串：

```
void remove(page_t addr)
```

删除一个字符串，这个字符串在文件中的位置是 addr

查询字符串：

```
void query(page_t addr, char* buf)
```

查询在文件中地址为 addr 的字符串，并把它拷贝到 buf 里面

比较字符串大小：

```
int compare(page_t addr, const char *str)
```

这个函数可以比较一个文件里的字符串和一个内存里的字符串的大小。如果文件里的字符串比较大返回 1，如果相等返回 0，否则返回-1

ii. 定长对象存储

在 Fix_Alloc 中，有和 page_t 对应的宏 addr_t，由于一个页中可能存在多个定长对象，所以 addr_t 表示的是一个对象在文件中的地址

构造函数：

```
Fix_Alloc(const char* filename, int single=0)
```

输入一个文件名，这个文件将作为定长对象的容器。每个对象大小为 single。

插入对象：

```
addr_t insert(void *item)
```

item 是一个泛型指针，指向一个大小为 single 的结构体。把这个对象写入到文件中，并返回它在文件中的位置。

删除对象：

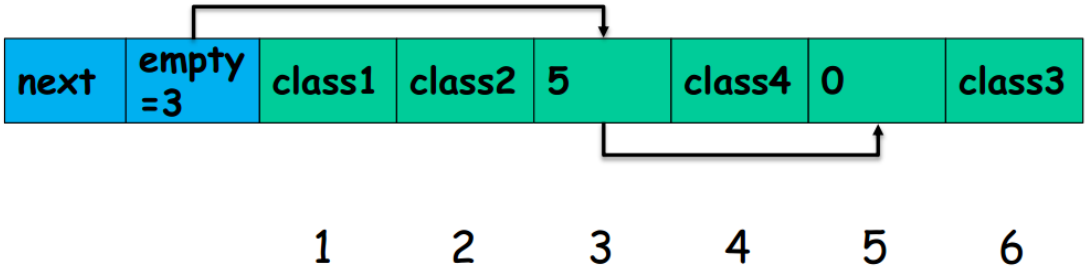
```
void remove(addr_t addr)
```
addr 是一个对象在文件中的地址，删除这个对象

使用对象：
```
void* use(addr_t addr)
```
addr 是一个对象在文件中的地址，把这个对象用 mmap 映射到内存中，并返回其虚存地址

释放对象：
```
void unuse(addr_t addr)
```
addr 是一个已经被映射到内存中的对象，现在要在虚存中释放这个对象。

iii. 空闲空间管理

由于存在删除操作，所以需要一个有效地数据结构管理所有的空白页和空白对象空间。这里仅介绍最复杂的 Fix_Alloc 的空闲空间管理。



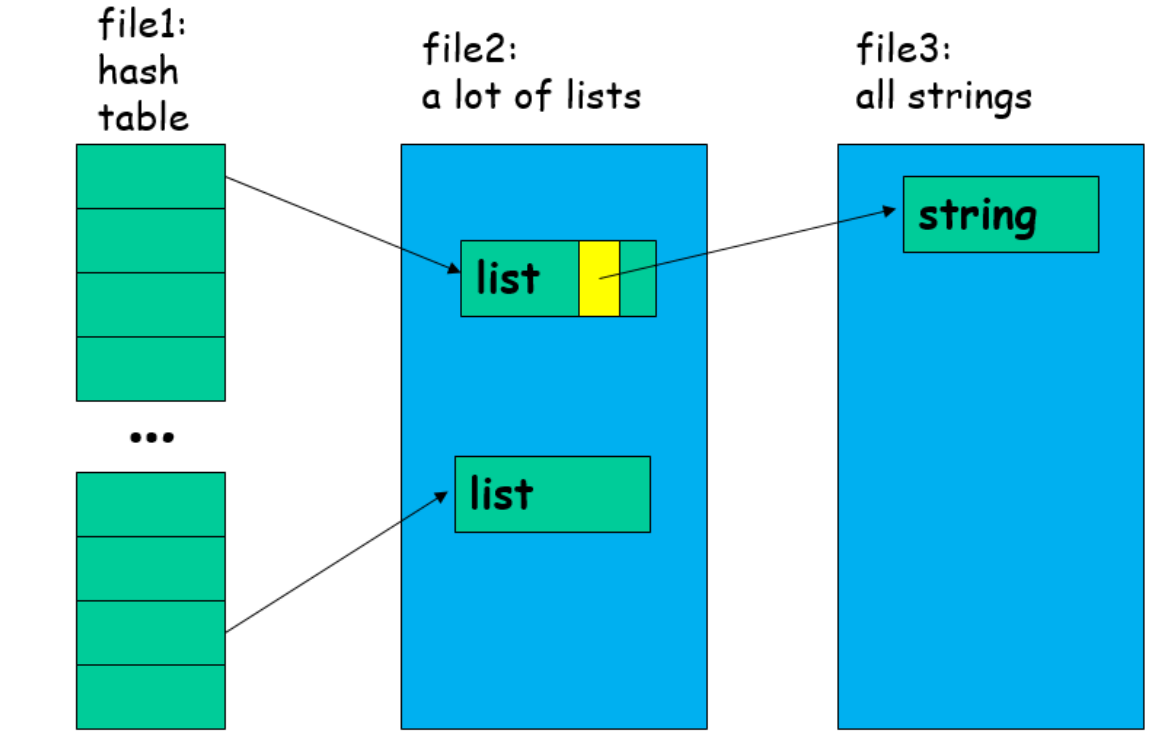
如图所示，表示一个对象存储文件中的一页。这一页可以放 6 个对象(在程序中对应变量 blockPerPage)。empty 是一个单向链表的头，这个单向链表链接了这一页中的所有空闲空间。如果这一页有空闲空间，那么这个 page 就会存在于一个全局的空闲链表中，而 next 指向的就是下一个有空闲空间的 page。下图是整个文件的 header：

整个文件的头部记录了全局空闲链表的头指针，即 Empty。

三、 哈希表实现

哈希表的本质就是一个数组，这个数组上每一个位置都指向一个链表。所以我先使用底层工具实现了一个 List 类，即链表类。由于底层操作都已经包装好，所以这个链表类实现起来很容易。

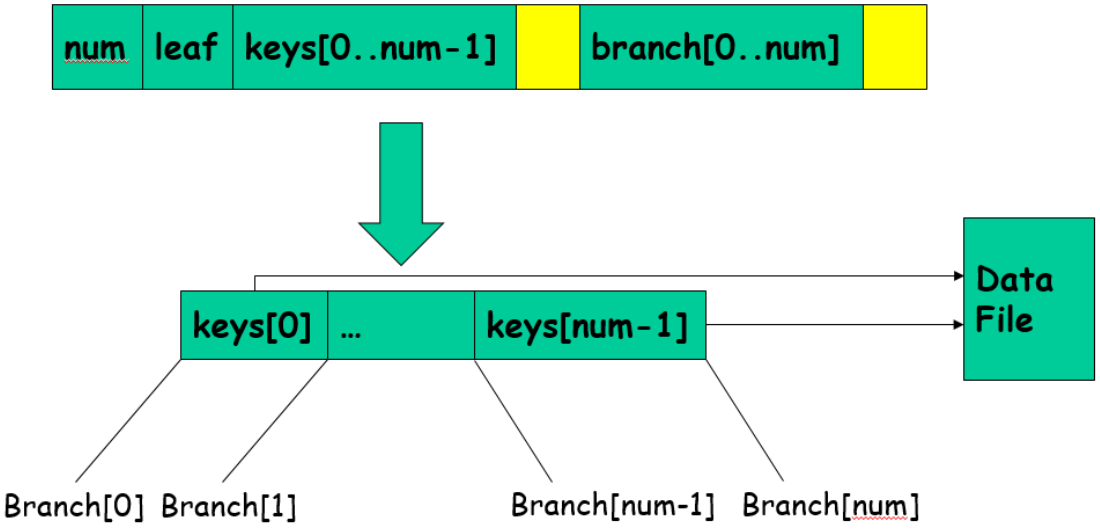
哈希表类 HashTable 的构造函数需要三个参数，即三个文件名。这三个文件的作用如下图所示：



第一个文件里存的是 1000000 个指针，其中有一些是-1 表示还没有对象被散列到这个值。其他的指针指向第二个文件中的连表头，链表中的某一项又会指向第三个文件中的一个字符串。这就是第一个存储系统的实现。

四、 B+树实现

B+树相对而言比较复杂，这里不赘述其原理。有一些实现细节如下：
我是用的是 510 叉 B+树，这样每个页正好可以存下一个节点的信息。树中每个节点存储两个数组。第一个数组是 keys，keys[i]表示该节点的第 i 个 key(字符串)在数据文件中的位置。第二个数组是 branch，branch[0..num]是指向儿子的指针。如下图所示：



五、 测试设计

我使用随机生成的测试数据进行测试，并写了使用 std::map 进行模拟的对拍程序 checker.cpp 进行对拍。测试数据的结构如下:

第一行：一个整数 N，表示有 N 个操作

接下来 N 行：每行可能是下面四种格式之一：

1 key value 表示插入(key, value)

2 key 表示删除 key

3 key value 表示修改(key, old)，变成(key, value)

4 key 表示查询 key 对应的 value

对于两个数据存储系统，我分别写了两个 main.cpp，来通过输入文件调用两个系统的类接口。

六、 测试结果

对于哈希表和 B+树两种数据结构的测试结果如下：

size	B+ Tree	Hash Table
100	0.033s	2.429s
1000	0.204s	2.724s
1000	0.052s	2.462s
10000	0.378s	2.523s
100000	2.565s	3.165s
500000	29.252s	14.787s
2000000	36min42s	19min25s

可见随着数据规模的增加，两个数据存储系统所需要的时间也会增加。可见数据不大的时候 B+树比较快，但是对于最后两组数据哈希表比较快。但是，时间并不能完全反应数据结构的特点。时间还和实现方式、编译优化、内核优化等等有关系。

由于我对于底层进行了包装，所以我可以很方便的测得两个系统分别在索引文件、数据文件中造成了多少次 page map。我认为这是体现数据结构特点更好的指标。

下面这张表所示的是两种数据结构访问索引文件的频率。可见在索引文件的访问上 B+树有着根本的优势，这也是多叉平衡树被发明的原因。

size	B+ Tree	Hash Table
100	31	1002007
1000	10	1002400

1000	17	1002400
10000	7739	1006274
100000	90304	1043295
500000	456584	1205931
2000000	976318	1912275

下面这张图所示的是两种数据结构访问数据文件的频率。可见 B+树对数据文件的访问非常频繁，要比哈希表高 10 倍左右。

size	B+ Tree	Hash Table
100	1402	1008
1000	16256	9224
1000	8512	2142
10000	118935	19556
100000	1530774	195468
500000	8814046	997681
1000000	42137424	5076071

七、 结果分析

这里主要分析 B+树为什么这么频繁地访问数据文件。

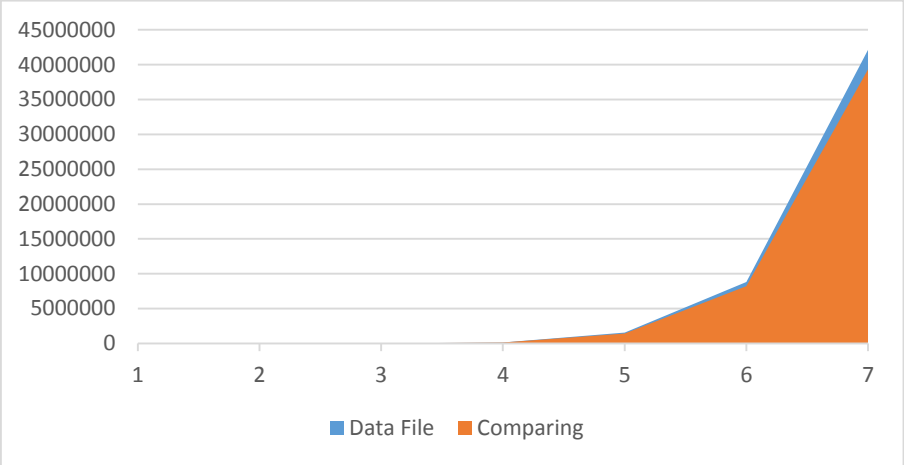
理论上，B+树有一个很大的缺点，那就是在单个节点内的查找会大大增加复杂度常数。如果使用链表，那么插入和查询的常数永远是 510。我使用数组实现的，所以可以使用二分查找降低常数。

根据理论，我猜想 B+树的性能瓶颈出现在键值比较上。所以我进行了如下测试：

	Data File	Comparing
trace0	1402	634
trace1	16256	9141
trace2	8512	7066
trace3	118935	106291
trace4	1530774	1407862
trace5	8814046	8205353
trace6	42137424	39336628

表中左面的数表示 B+数在数据文件中总共造成的 map 次数，右面的数表示 B+树

比较键值的次数。由于键值比较涉及一个内存字符串和一个硬盘字符串的比较，即 Random_Alloc 的 compare 方法，每次比较都至少会引发一次 mmap。形象地画出图表就是这样：

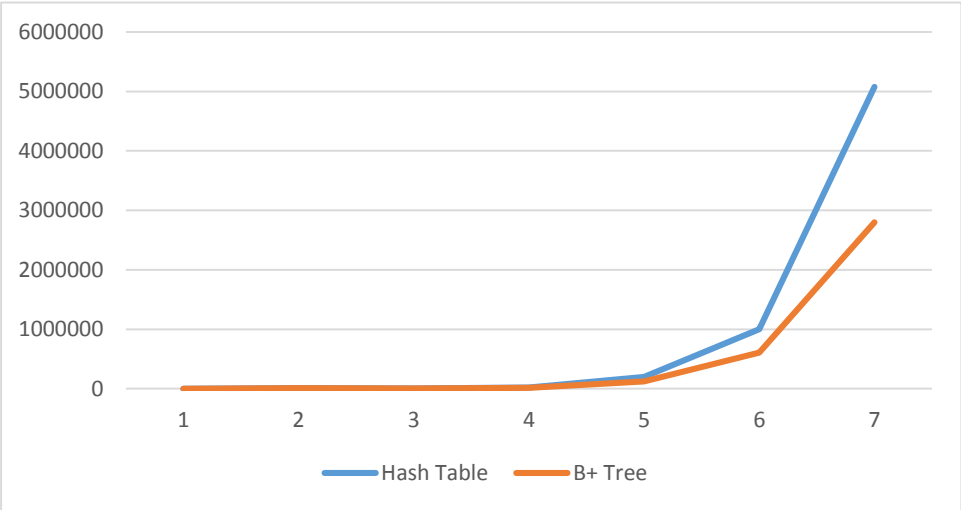


可见绝大多数 page map 都是键值比较造成的，这个和 B+树的理论缺陷相符合。

八、 扩展思考

既然 B+树在这个应用中不如哈希表，那么为什么要用 B+呢？

在现实数据库中，比如 mysql，一个 table 会有主键(Primary Key)，而且一般做法都是把一个 int 作为主键而且设为自增(Auto Increment)。我们注意到，如果用 int 作为索引，那么 B+树在进行键值比较的时候就不需要访问数据文件，效率也会大大提高。如果我们把 B+树进行比较的 page map 次数刨去，则会得到下面这张图：



预计如果用 int 作为键值，哈希表的复杂度也会相应降低，所以二者访问数据文件的时间就会持平。这个时候 B+树在索引文件上的优势就体现出来了。